

# Compilation Principle 编译原理

## 第14讲: 语义分析(4)

### 张献伟

#### <u>xianweiz.github.io</u>

DCS290, 4/15/2021





## Review Questions (1)

• How is Semantic Rules differing from Actions?

Rules are used in SDD, actions are for SDT. Actions are specifically placed at somewhere of the production body.

• What is S-SDD?

Synthesized-SDD, with only synthesized attributes.

- S-SDD is suitable for bottom-up or top-down parsing? Bottom-up. Natural to evaluate the parent after seeing all children.
- How to convert an S-SDD into SDT?
   Place each rule inside '{}' at the end of production.
- If implementing the SDT of S-SDD in LR parsing, when to execute the actions?

Along with reduction.



## Review Questions (2)

• Is the SDD a L-SDD?

$$A \rightarrow X Y Z$$
  $Y.i = f(Z.z, A.s)$ 

NO. Z is right to Y, A.s is synthesized attribute.

- Why do we prefer to do semantic analysis during parsing? Skip parse-tree generation, saving time and memory.
- For S-SDD in LR-parsing, how to change parse stack? Save synthesized attributes into the stack, along with state/symbol.
- How to convert L-SDD into SDT?

Inherited rules: place before the non-terminal; syn: production end.

• L-SDD can be implemented in LL- or LR-parsing?

Both. LL: predictive, recursive-descent; LR





## L-SDD in LL Parsing[非递归预测]

- Extend the parse stack to hold **actions** and certain **data items** needed for attribute evaluation[扩展语法分析栈]
  - Action-record[动作记录]: represent the actions to be executed
  - Synthesize-record[综合记录]: hold synthesized attributes for non-terminals
  - Typically, the data items are copies of attributes[属性备份]
- Manage attributes on the stack[管理属性信息]
  - The inherited attributes of a nonterminal A are placed in the <u>stack record</u> that represents that terminal[符号位放继承属性]
     Action-record to evaluate these attributes are immediately <u>above</u> A
  - The synthesized attributes of a nonterminal A are placed in a separate synthesize-record that is immediately <u>below</u> A[综合属 性另存放]



action	Code		
A	Inh Attr.		
A.syn	Syn Attr.		



## L-SDD in LL Parsing (cont.)

- Table-driven LL-parser
  - Mimics a leftmost derivation --> stack expansion
- A -> BC, suppose nonterminal C has an inherited attr C.i
  - C.i may depend not only on the inherited attr. of A, but on all the attrs of B
    - Extra care should be taken on the attribute values
  - Since SDD is L-attributed, surely that the values of the inherited attrs of A are available when A rises to stack top

Thus, available to be copied into C

 A's synthesized attrs remain on the stack, below B and C when expansion happens







## L-SDD in LL Parsing (cont.)

- A -> BC: C.*i* may depend not only on the inherited attr. of A, but on all the attrs of B
  - Thus, need to process B completely before C.i can be evaluated
  - Save temporary copies of all attrs needed by evaluate C.i in the action-record that evaluates C.i; otherwise, when the parser replaces A on top of the stack by BC, the inherited attrs of A will be gone, along with its stack record
  - 变量展开时(i.e., 变量本身的记录出栈时), 若其含有继承属性,则要将集成属性复制给后面的动作记录
  - -综合记录出栈时,要将综合属性值复制给后面的动作记录

action	Code
А	
A.syn	





## Example

(1) T -> F { T'.inh = F.val } T' { T.val = T'.syn }
(2) T' -> \* F { T<sub>1</sub>'.inh = T'.inh x F.val } T<sub>1</sub>' { T'.syn = T<sub>1</sub>'.syn }
(3) T' -> ε { T'.syn = T'.inh }
(4) F -> digit { F.val = digit.lexval }

Three kinds of symbols:

- 1) Terminal
- 2) Non-terminal
- 3) Action symbol

(1) T -> F { a <sub>1</sub> } T' { a <sub>2</sub> }	a <sub>1</sub> : T'. <i>inh</i> = F. <i>val</i>
	a <sub>2</sub> : T. <i>val =</i> T' <i>.syn</i>
(2) T' -> * F { a <sub>3</sub> } T <sub>1</sub> ' { a <sub>4</sub> }	$a_3$ : T <sub>1</sub> '.inh = T'.inh x F.val
	a <sub>4</sub> : T'.syn = T <sub>1</sub> '.syn
(3) T' -> ε { a <sub>5</sub> }	a <sub>5</sub> : T'. <i>syn</i> = T'. <i>inh</i>
(4) F -> digit { a <sub>6</sub> }	a <sub>6</sub> : F. <i>val =</i> digit. <i>lexval</i>



## Example (cont.)

(1) T -> F { a <sub>1</sub> } T' { a <sub>2</sub> }	a <sub>1</sub> : T'. <i>inh</i> = F. <i>val</i>
	a <sub>2</sub> : T. <i>val =</i> T'.syn
(2) T' -> * F { a <sub>3</sub> } T <sub>1</sub> ' { a <sub>4</sub> }	a <sub>3</sub> : T <sub>1</sub> '.inh = T'.inh x F.val
	a <sub>4</sub> : T'.syn = T <sub>1</sub> '.syn
(3) Τ' -> ε { a <sub>5</sub> }	a <sub>5</sub> : T'. <i>syn</i> = T'. <i>inh</i>
(4) F -> digit { a <sub>6</sub> }	a <sub>6</sub> : F. <i>val =</i> digit <i>.lexval</i>

Input: 3 \* 5 † †

Stack top 'digit' matches the input '3' - pop 'digit', but value copy is needed a<sub>6</sub>: stack[top-1].*val* = stack[top].*d\_lexval* 

	digit	{ a <sub>6</sub> }	Fsyn	{ a <sub>1</sub> }	T'	T'syn	{ a <sub>2</sub> }	Tsyn	\$
	lexv=3	d_lexv=3	val =3	val=3	inh	val		val	
-									

完整步骤见**←:** <u>MOOC:语法制导翻译-3</u>





## L-SDD in LR Parsing

- What we already learnt
  - LR > LL, w.r.t parsing power
    - We can do bottom-up every translation that we can do top-down
  - S-attributed SDD can be implemented in bottom-up way
    - All semantic actions are at the end of productions, i.e., triggered in reduce
- For L-attributed SDD on an LL grammar, can it be implemented during bottom-up parsing?
  - Problem: semantic actions can be in anywhere of the production body

(1)  $T \rightarrow F \{ T'.inh = F.val \} T' \{ T.val = T'.syn \}$ (2)  $T' \rightarrow F \{ T_1'.inh = T'.inh \times F.val \} T_1' \{ T'.syn = T_1'.syn \}$ (3)  $T' \rightarrow \epsilon \{ T'.syn = T'.inh \}$ (4)  $F \rightarrow digit \{ F.val = digit.lexval \}$ 





## The Problem

- It is not natural to evaluate inherited attributes
   Example: how to get T'.inh
- Claim: inherited attributes are on the stack
  - Left attributes guarantee they've already been computed
  - But computed by previous productions deep in the stack
- Solution
  - Hack the stack to dig out those values

(1)  $T \rightarrow F \{ T'.inh = F.val \} T' \{ T.val = T'.syn \}$ (2)  $T' \rightarrow F \{ T_1'.inh = T'.inh \times F.val \} T_1' \{ T'.syn = T_1'.syn \}$ (3)  $T' \rightarrow \varepsilon \{ T'.syn = T'.inh \}$ (4)  $F \rightarrow digit \{ F.val = digit.lexval \}$ 



\*

digi

digit

#### Marker

- Given the following SDD, where  $|\alpha| \neq |\beta|$ A -> X  $\alpha$  { Y.in = X.s } Y | X  $\beta$  { Y.in = X.s } Y Y ->  $\gamma$  { Y.s = f(Y.in) }
- Problem: cannot generate stack location for Y.in
  - Because X.s is at different relative stack locations from Y
- Solution: insert markers  $M_1$ ,  $M_2$  right before Y  $A \rightarrow X \alpha M_1 Y \mid X \beta M_2 Y$   $Y \rightarrow \gamma \{ Y.s = f(stack[top - |\gamma|].s) \} // Y.s = M_1.s \text{ or } Y.s = M_2.s$   $M_1 \rightarrow \epsilon \{ M_1.s = stack[top - |\alpha|].s \} // M_1.s = X.s$  $M_2 \rightarrow \epsilon \{ M_2.s = stack[top - |\beta|].s \} // M_2.s = X.s$



## Modify Grammar with Marker

- Given an L-SDD on an LL grammar, we can adapt the grammar to compute the same SDD during an LR parse
  - Introduce into the grammar a **marker nonterminal**[标记非终结 符] in place of each embedded action
    - $\square$  Each such place gets a distinct marker, and there is one production for any marker M, M ->  $\epsilon$  [空产生式]
  - Modify the action a if marker nonterminal M replaces it in some production A ->  $\alpha$  { a }  $\beta$ , and associate with M ->  $\epsilon$  an action a' that
    - Copies, as inherited attrs of M, any attrs of A or symbols of α that action a needs (e.g., M.i = A.i)
    - Computes attrs in the same way as a, but makes those attrs be synthesized attrs of M (e,.g., M.s = f(M.i))

A -> M B C M -> ε { M.*i* = A.*i*; M.*s* = f(M.*i*); }





#### Example





#### Stack Manipulation[栈操作]





## Semantic Analysis (4)

## Symbol Table





## Compilation Phases[编译阶段]

- Lexical analysis[词法分析]
  - Source code  $\rightarrow$  tokens
  - Detects inputs with illegal tokens
  - Is the input program lexically well-formed?
- Syntax analysis[语法分析]
  - Tokens  $\rightarrow$  parse tree or abstract syntax tree (AST)
  - Detects inputs with incorrect structure
  - Is the input program syntactically well-formed?
- Semantic analysis[语义分析]
  - AST  $\rightarrow$  (modified) AST + symbol table
  - Detects semantic errors (errors in meaning)
  - Does the input program has a well-defined meaning?





## Overview of Symbol Table

• Symbol table records info of each symbol name in a program[符号表记录每个符号的信息]

- symbol = name = identifier

- Symbol table is created in the semantic analysis phase[语 义分析阶段创建]
  - Because it is not until the semantic analysis phase that enough info is known about a name to describe it
- But, many compilers set up a table at lexical analysis time for the various variables in the program[词法分析阶段准备]
  - And fill in info about the symbol later during semantic analysis when more information about the variable is known
- Symbol table is used in code generation to output assembler directives of the appropriate size and type[后续代码生成阶段使用]



#### Variable[程序变量]

- What are **variables** in a program?
  - Variables are the names you give to computer memory locations which are used to store values in a computer program
  - Retrieve and update the variables using the names
- Variable declaration and definition[声明和定义]
  - Declaration: informs the compiler type and name of a variable[ 类型和名字]
  - Definition: to assign a memory[内存空间分配]
     Once we assign or initialized a value compiler allocates the memory



#### Example

```
1 #include <stdio.h>
        2
        3 int g_val;
        4
        5 int main() {
              int l_val;
        6
        7
              static int s_val;
        8
        9
              printf("q_val=%d, l_val=%d, s_val=%d\n", q_val, l_val, s_val);
       10
       11
              return 0;
       12 }
[xianwei@test>]$ gcc -Wall -g -o testc testc.c
testc.c:9:52: warning: variable 'l_val' is uninitialized when used here [-Wuninitialized]
   printf("g_val=%d, l_val=%d, s_val=%d\n", g_val, l_val, s_val);
                                                   ^~~~~
testc.c:6:13: note: initialize the variable 'l_val' to silence this warning
   int l_val;
            = 0
1 warning generated.
[xianwei@test>]$ ./testc
g_val=0, l_val=282353718, s_val=0
[xianwei@test>]$ ./testc
g_val=0, l_val=142671926, s_val=0
[xianwei@test>]$ ./testc
g_val=0, l_val=227987510, s_val=0
                                          19
山大岸
```

## Binding[绑定]

- Binding: match identifier use with definition[使用-定义]
  - Definition: associating an id with a memory location
  - Hence, binding associates an id use with a location
  - Binding is essential step before machine code generation
- If there are multiple definitions, which one to use?

```
void foo()
{
    char x; /* allocated at mem[0x100] */
    ...
    {
        int x; /* allocated at mem[0x200] */
    ...
    x = x + 1; /* add mem[0x100],1 ? add mem[0x200],1 ?
}
```



#### Scope[作用域]

- Scope: program region where a definition can be bound
  - Uses of identifier in the scope is bound to that definition
  - For C: auto/local, static, global
- Some properties of scopes
  - Use not in scope of any definition results in undefined error
  - Scopes for the same identifier can never overlap

There is at most one binding at any given time

Two types: <u>static scoping</u> and <u>dynamic scoping</u>
 Depending on how scopes are formed



#### Static Scoping[静态作用域]

- Scopes formed by where definitions are in program text[ 一个声明起作用的那段区域]
  - Also known as lexical scoping since related to program text C/C++, Java, Python, JavaScript [也叫词法作用域]
- Rule: bind to the closest enclosing definition





## Dynamic Scoping[动态作用域]

- Scopes formed by when definitions happen during runtime[运行时决定]
  - Perl, Bash, LISP, Scheme
- Rule: Bind to most recent definition in current execution

```
void foo()
{
  (1) char x;
  (2) if (...) {
  (3) int x;
  (4) ...
   }
  (5) x = x + 1;
  }
```

- Which *x*'s definition is the most recent?
  - Execution (a): ...<mark>(1)</mark>...(2)...(5)
  - Execution (b): ...(1)...(2)...(3)...(4)...(5)





## Static vs. Dynamic Scoping

- Most languages that started with dynamic scoping (LISP, Scheme, Perl) added static scoping afterwards
- Why? With dynamic scoping ...
  - All bindings are done at execution time
  - Hard to figure out by eyeballing, for both compiler and human
- Pros of static scoping[静态的好处]
  - Static scoping leads to fewer programmer errors
    - Bindings readily apparent from lexical structure of code
  - Static scoping leads to more efficient code
    - Compiler can determine bindings at compile time
    - Compiler can translate identifier directly to memory location
    - Results in generation of efficient code
- For this class, we will discuss static scoping only



## What is Symbol Table[符号表]

- **Symbol**: same thing as **identifier** (used interchangeably)
- Symbol table: a compiler data structure that tracks info about all program symbols
  - Each entry represents a definition of that identifier
  - Maintains list of definitions that reach current program point
  - List updated whenever <u>scopes</u> are entered or exited
  - Used to perform <u>binding</u> of identifier uses at current point
  - Built by either...
    - Traversing the parse tree in a separate pass after parsing
       Using semantic actions as an integral part of parsing pass
- Usually discarded after generating executable binary
  - Machine code instructions no longer contain symbols
  - For use in debuggers, symbol tables may be included
     To display symbol names instead of addresses in debuggers
     For GCC, using 'gcc -g ..." includes debug symbol tables



## Maintaining Symbol Table[维护]

• Basic idea

int x=0; ... void foo() { int x=0; ... x=x+1; } ... x=x+1 ...

- Before processing *foo*:
  - Add definition of x, overriding old definition of x if any
- After processing foo:
  - Remove definition of x, restoring old definition of x if any
- Operations
  - enter\_scope() start a new nested scope
  - exit\_scope() exit current scope
  - find\_symbol(x)
  - add\_symbol(x)
  - check\_symbol(x)

find the information about x add a symbol x to the symbol table true if x is defined in current scope



